symmetry

Article

MULBER: Effective Android Malware Clustering Using Evolutionary Feature Selection and Mahalanobis Distance Metric

Pradeepkumar Duraisamy Soundrapandian and Geetha Subbiah *

School of Computer Science and Engineering, Vellore Institute of Technology, Chennai 600127, Tamil Nadu, India
* Correspondence: geetha.s@vit.ac.in

Abstract: Symmetric and asymmetric patterns are fascinating phenomena that show a level of co-existence in mobile application behavior analyses. For example, static phenomena, such as information sharing through collaboration with known apps, is a good example of a symmetric model of communication, and app collusion, where apps collaborate dynamically with unknown malware apps, is an example of a serious threat with an asymmetric pattern. The symmetric nature of app collaboration can become vulnerable when a vulnerability called PendingIntent is exchanged during Inter-Component Communication (ICC). The PendingIntent (PI) vulnerability enables a flexible software model, where the PendingIntent creator app can temporarily share its own permissions and identity with the PendingIntent receiving app. The PendingIntent vulnerability does not require approval from the device user or Android OS to share the permissions and identity with other apps. This is called a PI leak, which can lead to malware attacks such as privilege escalation and component hijacking attacks. This vulnerability in the symmetric behavior of an application without validating an app's privileges dynamically leads to the asymmetric phenomena that can damage the robustness of an entire system. In this paper, we propose MULBER, a lightweight machine learning method for the detection of Android malware communications that enables a cybersecurity system to analyze multiple patterns and learn from them to help prevent similar attacks and respond to changing behavior. MULBER can help cybersecurity teams to be more proactive in preventing dynamic PI-based communication threats and responding to active attacks in real time. MULBER performs a static binary analysis on the APK file and gathers approximately 10,755 features, reducing it to 42 key features by grouping the permissions under the above-mentioned four categories. Finally, MULBER learns from these multivariate features using evolutionary feature selection and the Mahalanobis distance metric and classifies them as either benign or malware apps. In an evaluation of 22,638 malware samples from recent Android APK malware databases such as Drebin and CICMalDroid-2020, MULBER outperformed others by clustering applications based on the Mahalanobis distance metric and detected 95.69% of malware with few false alarms and the explanations provided for each detection revealed the relevant properties of the detected malware.

Keywords: intent; pending intent; intent analysis; binary analysis; Mahalanobis distance metric; evolutionary feature selection

check for updates

Citation: Duraisamy Soundrapandian, P.; Subbiah, G. MULBER: Effective Android Malware Clustering Using Evolutionary Feature Selection and Mahalanobis Distance Metric. *Symmetry* **2022**, *14*, 2221. https://doi.org/10.3390/sym14102221

Academic Editor: Alexander Shelupanov

Received: 10 September 2022
Accepted: 8 October 2022
Published: 21 October 2022

1. Introduction

Android is one of the most popular platforms for smartphones with an approximate market size of 71% [1]. This attracts hackers to target the platform with malicious applications. Android provides more flexibility than other platforms such as the ability to install third-party apps from outside the Google Play Store. This bypasses the security measures provided by the Google Play Store and therefore malware can be installed from unverified sources. In 2021, Kaspersky found 3,464,756 malicious installation packages, where 97,661 were new mobile banking Trojans and 17,372 were new mobile ransomware Trojans [2].

The Android sandbox and the Android permission system prevents a malware application from accessing sensitive information without the involvement of other applications, that is, an application has to explicitly request permission from the user during installation or at runtime. Though many users tend to grant the explicit permissions without understanding the consequences, we identify these permission-granting mechanisms as a controllable model. For example, at any point in time, a user can grant or revoke the requested permissions of an app. On the other hand, apps can dynamically collaborate with each other by exchanging Intents or PendingIntents.

A static analysis of an application extracts various important features from an app and helps to prevent a malware from installing on a device. The static analysis helps to identify the symmetric behavior of an application by traversing the app's communication path. With Android, applications communicate primarily by exchanging Intents among components [3]. The Intents carry the action and supporting data to perform the task by other receiving components. By statically analyzing this component relationship, the system can determine the app's dynamic behavior. This shows the symmetric pattern of an application. However, a static analysis cannot capture every symmetric pattern completely when an app is using implicit Intent to collaborate. Implicit Intent is a kind of open broadcasting where any app can participate to receive the Intent exchanged through that channel. Such a scenario indirectly brings an asymmetric nature into an app at runtime. This runtime asymmetric behavior of an application may lead to potential security breaches or risks that can leak sensitive information when a malware app receives the implicit Intent.

Some of the well-known research by Kirin [4], Stowaway [5], and RiskRanker [6] introduces only a small runtime overhead. In contrast, the combination of static and dynamic analyses introduces a large overhead but is effective in malicious activity. Most of the existing research works are built on pre-defined manually crafted patterns, which are limited to new malware apps that follow different attack patterns.

A leaky communication pattern—is one where an app exposes the PendingIntent vulnerability or communicates with apps without compatible permissions. Each Android app has distinctive behavioral properties such as communication patterns and permission usage. A benign app may not have a leaky communication pattern, whereas a vulnerable app may have such patterns. Recently, such PendingIntent leaks have been identified and reported on CVE forums [7–10].

Understanding these behavioral patterns can help to improve app security in several ways. If there is an X $\longrightarrow$ Y communication pattern:

1. Both X and Y can be mapped to the same behavioral set so that if X is vulnerable, then Y would become consequent, and vice versa.
2. Vulnerability validation could be applied to just one app out of the two.
3. Vulnerability in X could be targeted at apps that communicate with Y.
4. X and Y could be combined into a new behavior model, for example, Y's vulnerability leads to vulnerability in X, (i.e., Y $\longrightarrow$ X).

Contributions—In this paper, we present MULBER, a lightweight method that infers app communication patterns automatically and identifies malware using the Mahalanobis distance metric [11].

The Mahalanobis distance metric is an effective multivariate distance metric, where the distance is calculated between a point and a distribution. In general, Euclidean space [12] is used to describe the multivariate analysis through a coordinate (x-axis and y-axis) system. However, in real time, an app is represented by multiple features, where the importance between features varies according to the participants and scenarios. In such a case, classifying an app based on Euclidean distance may lead to false positives. Representing more than two variables in a planar coordinate is difficult and can reduce the efficiency of a classification in a case where some variables are not considered in the classification. On the other hand, the Mahalanobis distance metric calculates the distance by finding the covariance associated with all the variables together.

MULBER performs a broad static analysis, gathering app communication patterns, app permissions, app PendingIntent leaks, and their exposed components. The features are organized as a *topological space*. For example, an application communicating by exposing PendingIntent is mapped to a region bound by the app's permissions and components. For a given topological space (X, τ), a subset B of X is bound in X. Similarly, the subspace topology on B is defined by

$$\tau_B = \{B \cap U \mid U \in \tau\}. \tag{1}$$

This representation of features and its association with other features as a topological space enables MULBER to relate the patterns of features indicative for malware automatically using the Mahalanobis distance metric. The extracted features provide an application's dynamic behavior and thus can provide useful insights to predict the malware communication pattern from an application.

From our experiments on 22,638 applications from different markets, MULBER classified applications based on their vulnerable communication patterns. On average, the analysis of an application takes less than 86 s to predict on a regular computer for an average file size of 18 MB. To the best of our knowledge, MULBER is the first method that provides insights based on malware communication patterns using the Intents and PendingIntents communication patterns of an Android application.

In summary, this paper makes the following contributions to the detection of Android malware:

1. We introduce a method combining binary analysis and the Mahalanobis distance metric based on a clustering method that is capable of identifying Android malware with high accuracy and few false alarms.
2. MULBER can analyze an application's communication patterns based on Intent and PendingIntent exchanges. However, MULBER cannot analyze the obfuscated or dynamically loaded malware, loading from the resources folder, or exporting sensitive information to malware servers.

We summarize our method using the following two topics:

Feature Extraction Tool—We implement a feature extraction tool that can automatically collect the communication patterns from an application as evidence and present them as a .csv (comma separated values) file, and extract the important features using an evolutionary feature selection method.
Mahalanobis Distance-Based Classifier—We use the Mahalanobis distance metric instead of the Euclidean metric to classify an app as benign or malware.

The rest of this paper is organized as follows. The related works are discussed in Section 2. MULBER and its detection methodology are introduced in Section 3. The experiments and comparisons with related approaches are presented in Section 4. The limitations of MULBER are discussed in Section 5. Finally, Section 7 concludes the paper.

2. Related Works

RAICC [13] and PIAnalyzer [14] studied the security threats of PendingIntent. RAICC instrumented the PendingIntent calls (AICC calls) by adding a method called startActivity() with the right Intent as a parameter, thereby converting the calls to standard ICC calls that could be further processed with IccTA [15] and Amandroid [16]. Amandroid [16] statically extracted the inter-component dependencies using control and dataflows between app components. The research by the authors explored PendingIntent-based security and created a novel framework based on ownership types to encapsulate the PendingIntent object dynamically [17]. PITracker [18] analyzed the Android APK to discover the PendingIntent vulnerabilities.

Summary of Android PendingIntent Study: Applications' PendingIntent data was extracted statically by RAICC and PIAnalyzer. However, the capability of an application (authorized by the user at runtime) to function as a prospective recipient of PI depends on additional dynamic features. Since such collusion cannot be foreseen statically, this is handled by dynamically checking the recipient component's privileges [19,20].

Static Analysis: Static analyzing and dynamic monitoring helps to prevent ICC attacks [21]. SEALANT [21] combined the static analysis of an app to extract the potential vulnerabilities and additional runtime monitoring of inter-app communications that utilized the information extracted through the static analysis to prevent ICC attacks. SALMA [22] used an incremental machine learning model to analyze app security. Barros et al. [23] used security annotations on the Intent data to track the taint flow. DINA [24] augmented an app's control and dataflow graphs by dynamically loading the classes and this was also able to resolve reflective calls. PREV framework [25] used the syntactic similarities of an app's functional descriptions and grouped them into clusters with permission redelegation [26] behaviors (reachable, privileged APIs). This knowledge base was then used for classifying new apps. FlowDroid [27] presented a static taint-analysis tool that could reduce false alarms by detecting context-based data leaks. ComDroid [28] detected Intent-based attacks such as Intent spoofing, UIR, and Intent vulnerabilities by performing a flow-sensitive and intra-procedural static analysis. CHEX [29] mapped the leaks of private data or component vulnerabilities as a dataflow problem, thereby identifying the potential risks based on the existence of flow patterns.

Based on their functional descriptions, AnFlo [30] divided programs into malware and benign categories, allowing for the identification of unusual sensitive dataflows. In order to safeguard user privacy and security when an app accesses sensitive information, it is crucial that the app handle the data safely and adhere to secure coding requirements. AnFlo used NLP to extract topics from Play Store-declared features. AnFlo used static taint analysis to extract the sensitive information flows from the app descriptions once the topics had been extracted. Finally, trustworthy applications were utilized to create the sensitive information flow models, which were then applied for the categorization of new apps. The security flaw in the unique permission model used by Android is compiled by TERMINATOR [31], which also records the timing of occurrences. TERMINATOR compiles the chronological order, the timing of an event, and the security flaws in the unique permission model used by Android. When the system is in a safe condition, the enforcer of TERMINATOR gives programs a limited set of permissions; when the system enters an unsafe state, the enforcer revokes the permissions.

Summary of Android Static Analysis: By communicating Intents across components, ICC can be established. In contrast to PendingIntent, where the identity and privilege of the PI originator are revealed with the receiver, Intent communication allows the receiver to sniff sensitive information. According to the results of the afore-mentioned study, SALMA [22] can be used to gradually examine the PI flow among components because it continuously monitors and analyzes component communication while also updating or removing policies in accordance with app permissions. Similar to this, we discovered that Amandroid [16] allowed us to generate a static single assignment (short SSA) for PI flow, forming an event graph (similar to TERMINATOR [31]) to separate malicious software from benign software.

Policies, Permissions and Intent-Based Filters: Kirin [4] detected the malicious behavior of an app at install time using kirin security rules. DREBIN [32,33] dynamically inferred the security patterns of malware applications rather than capturing the pattern manually similar to [4]. TaintDroid [34], is a well known data-leak detection tool that can extract fine-grained private information from Android apps and determine how it is actually used. DroidCap [35] used binder handles and facilitated privilege separation among an app's components through the compartmentalization of components into a logical app with a subset of privileges. Maxoid [36] confined the receiver from leaking information by creating multiple views of the initiator's state. Here, the initiator has the power to selectively commit or discard the updates from the delegates on its private state or public state, thereby preventing unwanted modifications by delegates. Aquifer [37] defined UI configurable workflow policies that can be configured by each participating application. DyPoldroid [38] proposes a novel methodology to dynamically analyze Android overly privileged applications that can misuse the permissions granted to it, called as Permission Abuse applications(shortly, PA-Apps). The DyPoldroid is a Android Enterprise based solution, that dynamically monitors the BYOD device for such PA-Apps and applies security policies at runtime to prevent such applications from installing to the device or revoke privileged permissions granted to the PA-Apps by the device owner, so that the device is protected dynamically from PA-Apps.

Summary of Policies, Permissions and Intent-Based Filters: Our empirical study on PendingIntent security used a static analysis methodology similar to DREBIN [32] to infer permission combinations and feature patterns from the code and manifest files of the application. Applications must collaborate to accomplish a number of shared objectives. The security model proposed by Maxoid [36] matched our research findings. For mobile applications that call other applications, Maxoid offers confidentiality and integrity. Maxoid restricts delegates from leaking or passing on their private or public state to other applications while allowing delegates (also known as receiver apps) to access or change the initiator (also known as sender) apps. In addition, the initiator has the option to selectively commit or delete updates made by delegates to avoid unauthorized changes. However, the initiator is unable to read or write the private state of its delegates. This research suggests security viewpoints for the ICC components; however, it is not possible to fully apply this model to PendingIntent security due to the selective committing or discarding of actions conducted by delegates, which is challenging given that the PendingIntent vulnerability can still be used even if the initiator is terminated.

3. Methodology

The MULBER model is shown in Figure 1. MULBER takes the Android APK as input and first performs a binary analysis of the Android APK to extract the feature sets and map them to a bounded vector space. Later, it performs a priori feature selection to identify the important features based on their frequency and feature co-occurrence, followed by feature selection, where MULBER calculates the feature distance using the Mahalanobis algorithm. Finally, MULBER extracts the evaluation metrics of the Mahalanobis-distanced data. The process is outlined as follows:

1. Binary analysis. In the first step, MULBER statically inspects the given Android APK and extracts the different feature sets from the application's manifest and dex code.
2. The extracted feature sets are then pre-processed, for example, handling null values and the standardization of the extracted data.
3. Key feature selection. Using an a priori algorithm, MULBER extracts the key features that can form a contour.
4. Modifying dataset. MULBER creates a new dataset using the selected features, thereby removing the other features.
5. Learning-based detection. The feature vector space enables us to identify malware clusters based on the Mahalanobis distance metric.
6. Explanation. In the last step, the features contributing to the detection of communication leaks in a malicious application are identified and presented to the user, explaining the possibility of future malicious communication patterns.

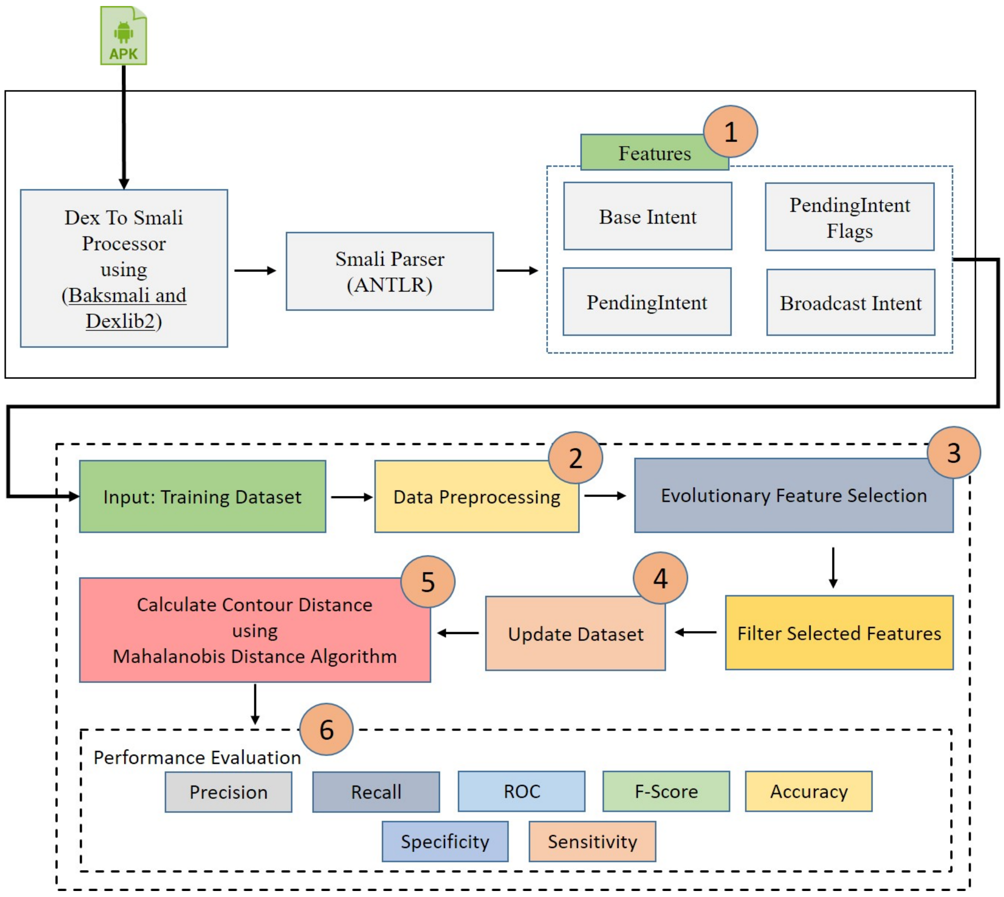


Figure 1. Overall process of MULBER model.

3.1. Binary Analysis

3.1.1. Features from Manifest.xml

MULBER initially processes the Manifest.xml and extracts the following information that enables our model to create a bounded communication model. We obtain the following data from the manifest. A file named AndroidManifest.xml, which contains information supporting the installation and subsequent execution of the program, must be included in every Android application. The details extracted from the Manifest.xml file are as follows:

- Requested permissions: The permission system is one of the most significant security features of Android. A user must actively authorize the permissions for an application to access security-related resources at installation time. Malicious software frequently requests more permissions than legitimate apps [4].
- Filtered intents: Inter-process and intra-process communications in Android are mainly performed through asynchronous messages called Intents. The Intents carry information and actions to be performed by the receiving component. We collect all Intents listed in the manifest as another feature set and these Intents show the communication patterns such as inbound and outbound links.

3.1.2. Features from DEX Code

Android applications are developed in Java and compiled into an optimized bytecode (dex) for the Dalvik virtual machine. By disassembling this bytecode, we can obtain information about the Intent/PendingIntent flow between components. We use this information to construct the following feature sets.

In general, app collaboration occurs by apps exchanging Intents with each other. This process of collaboration is called inter-component communication (ICC). We can classify ICC into two types—as unprotected implicit communication or protected explicit communication. Explicit Intent communication is safe since it confines the app to communicate only with a known component (i.e., statically fixing the communication channel between components). On the other hand, implicit Intent communication (also known as public broadcast) facilitates easy collaboration by not limiting the communication channel statically, thereby enabling easy collaboration with any other third-party apps; however, this flexibility comes with the additional risk of privacy leaks [28] or privilege leaks [14].

- Intent Communication: Android apps collaborate and share information with each other by exchanging a messaging object called an Intent [39]. An Intent is an unprotected object that carries data and the operation to be performed by the receiving application. Intents are used to start an activity, start a service, and to deliver a broadcast. The Android system classifies Intents into two types: (1) explicit Intents, and (2) implicit Intents.
  Explicit Intents specify the receiving application details such as the target app's package name or a fully-qualified component class name. In other words, explicit Intents are used to start a component in a known application because the class name of the activity or service is fixed at the Intent creation time itself. On the other hand, implicit Intents declare a general action to perform. The Android framework takes the liberty to identify the target component for this action, i.e., a component from another app can also handle it. For example, in the following code snippet, the source application is dynamically collaborating with the camera app through the implicit Intent. In Line 1, the implicit Intent is created, in Line 4, the file location is added to the Intent, and in Line 5, the implicit Intent is shared publicly. In Figure 2 we can see the possible application communication based on both the implicit and explicit communication mechanisms. By inspecting the communication patterns and the payload exchanged with other applications, we can capture the vulnerability of an app.

```
1   Intent implicitIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
2   File filename = new File(Environment.getExternalStorageDirectory(),  "image.jpg");
3   Uri photoURI = FileProvider.getUriForFile(this, "com.example.android.fileprovider",
    ↪  filename);
4   implicitIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
5   startActivityForResult(implicitIntent, IMAGE_REQUEST_ID);
```

- PendingIntent Communication: On the other hand, dynamic collaboration is also possible by exchanging PendingIntents (also known as Wrapping Intents (WI) [14]) between applications. The PendingIntent (PI) [40] dynamically grants authority that a PI creating app owns to the PI receiving app. This is a kind of temporary permission sharing between applications that lasts as long as the shared PendingIntent is valid. In the below code, the Wrapping Intent carries the created PendingIntent object between applications. In Line 1, we create an implicit Intent, and in Line 3, a PendingIntent object is created using the baseintent object created in Line 2, and in Line 4, the PendingIntent is parceled inside the implicit Intent. Finally, in Line 5, the PendingIntent is exported publicly.

```
1   Intent implicitIntent = new Intent("com.listener.action");
2   Intent baseintent = new Intent();
3   PendingIntent pendingIntent = PendingIntent.getActivity(this, 1, baseintent,
    ↪  PendingIntent.FLAG_UPDATE_CURRENT);
4   implicitIntent.putExtra("WI", pendingIntent);
5   sendBroadcast(implicitIntent);
```

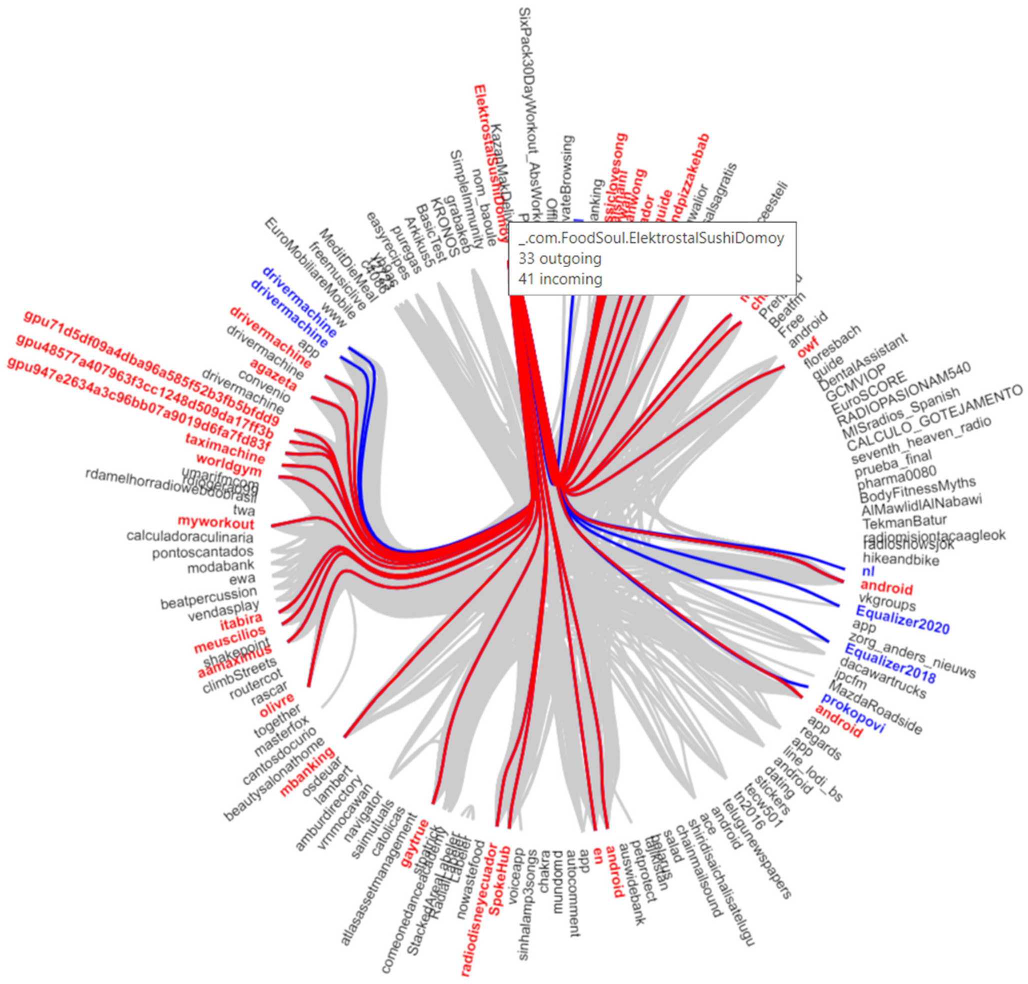


Figure 2. ICC Communication patterns between apps (inbound and outbound)

3.2. Android Permissions

The Android system provides 136 permissions under three broad categories, namely, normal permissions, signature permissions, and dangerous permissions. Each permission type indicates the scope of the restricted data that an app can access when the system grants that permission. The normal permissions and the signature permissions together are called the install-time permissions, and the dangerous permissions are called the runtime permissions.

Install-Time Permissions: The install-time permissions are the permissions that are automatically granted by the system when an app declares them in the app Manifest.xml at the app installation time. These permissions include normal and signature permissions. Android by default allows "normal and signature" permissions requested in the Android-Manifest.xml such as giving apps access to the internet. Similarly, the signature permissions are granted by default if the requesting application is signed with the same certificate as the application that declared the permission [41]. Android provides 46 normal permissions and 49 signature permissions that can be used by an application.

Runtime Permissions: The runtime permissions, also known as "dangerous" permissions, permit an app to access restricted data. Since the runtime permissions access private user data, these permissions require an app to request explicitly at runtime. For example, to access the device location, an app should explicitly request "ACCESS_FINE_LOCATION" and "ACCESS_COARSE_LOCATION" permissions, and to access the user contacts, an app should request "READ_CONTACTS" permissions. Android provides 41 dangerous permissions that can be used by an app at runtime. Actually, the dangerous permissions are not inherently dangerous but are vulnerable to attacks.

3.3. Security Features

MULBER collects almost every possible property from an APK that is classified as a static or dynamic security feature. MULBER classifies the entire dataset into two key features: (1) static features of an application, and (2) dynamic features of an application. The static features of an application are collected from the Manifest.xml and the dynamic features of an application are collected based on the usage of Intent and PendingIntent in the APK binary. MULBER totally collects 172 properties containing 21 static features and 15 PI and Intent-based dynamic features, and the app's usage of Android's 136 permissions is collected from three categories, which are normal, signature, and dangerous permissions. Each android permission is assigned an initial weight of 1. Table 1 displays the static and dynamic features collected from a sample Android application (considering the app privacy, we have purposefully hidden a few fields).

Manual Inspection of the App Features: Based on our knowledge and manual inspection of various PI-vulnerable apps, we found that the following static and dynamic features were meaningful as they directly represented the PendingIntent and Intent vulnerabilities. For example, features such as usage of dangerous permissions, unsafePICreation, unsafeBroadcast, and unsafePITransfer represented the vulnerable nature of the app rather than the other features.

Reasoning: From the above manually identified features, we could address the following three key issues:

1. Whether the investigated application used dangerous permissions.
2. Whether the investigated application had a vulnerable PI creation, for example, creation of a PI with an empty baseintent.
3. Whether the investigated application exposed the vulnerable PI using an implicit broadcast.

However, in order to automate the analysis, in MULBER, rather than identifying the key features manually, we follow an evolutionary feature selection method to identify the best-fit features that can be used to train our underlying machine learning model. MULBER extracts the features directly from the Android dex code. The smali/baksmali [42,43] is an assembler/disassembler for the dex format used by Dalvik, Android's Java VM implementation, using Dexlib2 [44]. The process of extracting the PendingIntent and the Intent communication patterns as features from the APK dex code is given in Algorithm 1.

Algorithm 1 Dataset Generation from Android APK

```
Data: APK Dataset
Result: CSV Dataframe with Multiple Features Extracted from APK
while APK.HasNext() do
    ENSURE—APK is Parseable
    if YES then
        SMALI := DexToSmali()
        Parse(SMALI)
        Features: ExtractFeatures(AST Node):
        if Node typeof PendingIntent then
            BaseIntent := Node.BaseIntent
            Flags := Node.Flags
        end
        if Node typeof Intent then
            PendingIntent := Node.Payload
            BroadcastType := Node.isPublic
        end
        CSVGenerator(Features)
    end
end
```

Table 1. The static and dynamic features collected from the APK using MULBER.

Static Features	**Sample Values**
Checksum	d17d80f58b36ebe71dec80fa5ae8542c
appPackageName	-
appName2	-
appSize(KB)	40,003
TotalPermissions	41
minSDKVersion	15
targetSDKVersion	26
TotActivities	1112
TotServices	40
TotBroadcastReceivers	51
TotExposedActivities	86
TotExposedServices	13
TotExposedBroadcastReceivers	57
OutBroadcastTags	android.intent.action.SENDTO android.intent.action.SEND
InBroadcastTags	android.intent.action.BOOT_COMPLETED android.net.wifi.WIFI_STATE_CHANGED androidnet.conn.CONNECTIVITY_CHANGE android.intent.action.PHONE_STATE android.intent.action.PACKAGE_REMOVED
owner	-
issuer	-
serialnumber	4e5b9304
algorithm	SHAlwithRSA (weak)
bit	1024-bit RSA key (weak)
version	3
Dynamic Features	**Sample Values**
noOfClz	22,347
noOfMethods	89,943
no0fIntents	520
noOfPendingIntents	37
FLAG_ONE_SHOT	0
FLAG_NO_CREATE	0
FLAG_CANCEL_CURRENT	1
FLAG_UPDATE_CURRENT	30
FLAG_IMMUTABLE	0
unsafePICreation	2
unsafeBroadcast	2
unsafePITransfer	0
NoOfProtectedBroadcast	2
UnsafeProtectedBroadcast	0
TotalBroadcast	48

3.4. Evolutionary Feature Selection

Feature selection is one of the key techniques in machine learning. MULBER uses an evolutionary feature selection algorithm to mine the frequent patterns from the given input dataset rather than follow a predefined feature set. The evolutionary algorithm is a generic optimization technique based on the ideas of natural evolution. The initialization phase starts with the parents creating the offspring using the crossover technique. The individuals in a subset are randomly generated from the population. As a rule of thumb, the crossover requires the attributes to be present at between 5% and 30% of the number of attributes as population sizes. Below 5% will affect the crossover mechanism. MULBER recursively considers all the parents from the given population in at least one crossover. A crossover helps to create large jumps in the fitness population landscape, thereby helping to fit better with multi-modal fitness landscapes, and is not associated with local extreme conditions.

Secondly, the created individuals will undergo a random mutation with a low probability, whereby a random state is added or removed from the selected gene. This helps our model to find subsets of features without creating a brute-force model of checking all possible combinations of our features. Thus, by using a multi-objective fitness function, we can find new solutions that optimize the cross-validation score while minimizing the selection of features.

Finally, the survival features are defined based on the fittest individuals from the given dataset, and the process of selecting the fittest individuals is called the selection method.

The evolutionary feature selection algorithm automates the process of selecting the best fit features from 172 features. Following the feature selection process, we use K-means clustering with the Mahalanobis distance metric method to cluster and partition the data into clusters. Since the objective of this paper is to discover the efficiency of applying the Mahalanobis metric rather than using the Euclidean metric, we have chosen to test our application using the vanilla clustering method (K-means) rather than some of the other advanced clustering methods such as fuzzy clustering [45], and RBF-based clustering [46], and using some other machine learning models like deep random forest [47], densenet-based deep learning [48].

3.5. Mahalanobis Distance Metric

The metrics for classifying Android apps as benign or malware involve relative mapping rather than discrete. For example, Euclidean distance [49] assumes all the dimensions (features) have the same unit of measurement. This makes the underlying system give equal importance to all the features. However, in Android apps, we know that all the features cannot have the same unit of measurement. For example, apps with dangerous permissions and PI leaks have different weights than apps with PI leaks and normal permissions. In such a scenario, we can apply the Mahalanobis distance metric, which transforms the vector into a zero mean vector (by subtracting the mean of each column from that column) with an identity matrix for covariance. Once the above process is completed, the Euclidean distance can be applied to the multivariate data.

MULBER classifies apps based on the permissions an app requests in its Manifest.xml. Similarly, apps with normal permissions are classified as *benign* and apps that request dangerous permissions are classified as *cynical*. Cynical apps are ones that should not be trusted initially without first performing a deep binary analysis of the application's behavior.

MULBER maps each application as a point in the metric space, where all the closely related apps form a contour. A metric is a non-negative function between two points x and y that describes the 'distance' between these two points. From a geometric point of view, the Euclidean distance between two points is the shortest possible distance between them. However, the Euclidean distance measure does not take the correlation between apps that are connected to other apps forming a contour graph. More clearly, the Euclidean distance is a distance calculated between two app points only and it does not consider how the rest of the correlated points vary.

An alternative approach is to scale the contribution of individual apps to the distance value according to the variability in the characteristics of each app in the communication network. This approach is considered by the Mahalanobis distance metric, which was developed by PC Mahalanobis [50,51]. The approach differs from Euclidean distance in the way that it takes into account the correlations between variables. The Mahalanobis distance metric is an effective distance metric that finds the distance between a point and a distribution.

$$D(X,\mu) = \sqrt{(X-\mu)'C^{-1}(X-\mu)} \tag{2}$$

Basic Definitions:

- Points—Points represents the variables that are mapped to an application and its characteristics, say, (x_1, y_1) .. (x_n, y_n).
- Mean—Represents the average of the given set of points (x_i, y_i), where i = 0...n.

$$(\mu_x, \mu_y) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i, \frac{1}{n} \sum_{i=1}^{n} y_i \right) \tag{3}$$

- Variance—The variance is a measurement of how dispersed the distribution of the provided collection of points is. It shows how the variable's distribution looks close to the mean value. A small variance implies a distribution of the supplied variable that is close to the mean value, whereas a large variance suggests a distribution of the random variable that is distant from the mean value. Given X and Y are separate random variables:

$$var(x) = \frac{1}{n} \sum_{i=1}^{n} (x_i, \mu_x)^2 \tag{4}$$

$$var(y) = \frac{1}{n} \sum_{i=1}^{n} (y_i, \mu_y)^2 \tag{5}$$

$$var(x + y) = var(x) + var(y) \tag{6}$$

- Covariance—Covariance is the directional relationship between the two variables; positive covariance means they move in the same direction, negative covariance means they move in an inverse direction, and zero covariance means there is no variation. The covariance metric compares the combined changes of the two variables, i.e., compares the variance rather than the dependency or strength between the given two variables.

$$cov(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i, \mu_x)(y_i, \mu_y) \tag{7}$$

- Covariance Matrix—The covariance matrix (also known as the variance–covariance matrix) represents the variance between the pair of elements along the diagonal and the covariance along the off-side of the diagonal. The covariance matrix gives the structured relationship between various variables in the given dataset.

$$C = \begin{bmatrix} var(x) & cov(x, y) \\ cov(x, y) & var(y) \end{bmatrix} \tag{8}$$

The Mahalanobis distance metric calculates the distance between two points in a multivariate space (i.e., correlated space). The Mahalanobis distance metric calculates the distance by taking the correlation values of the dataset elements, which is calculated using the covariance matrix. In Figure 3, we have selected a target point A (x1,y1) marked as X, which we need to fit into the correct cluster.

$$EuclideanDistance(p, q) = \sqrt{(p1 - q1)^2 + .. + (pn - qn)^2} \tag{9}$$

$$\begin{aligned} mB &= mean(benign) \\ mM &= mean(malware) \\ d1 &= db(mB, A) \\ d2 &= db(mM, A) \end{aligned} \tag{10}$$

$$calculateLessEculideanDistance(d1, d2) = (d1, Benign) \tag{11}$$

The Euclidean distance (Equation (9)) from the centroid of the benign cluster to point A is given by d1, and the Euclidean distance from the centroid of the malware cluster to point

A is given by d2 (given in Equation (12)). The Euclidean distance (d2) from the malware is higher compared to the Euclidean distance from the benign cluster, calculated using the above Equation (11). Figure 4 displays the results for six sample records; three were taken from each of the malware and benign datasets at random and tested for Euclidean distance. The results show that only 50% of the prediction was correct (i.e., the malware dataset samples were wrongly mapped to the benign cluster).

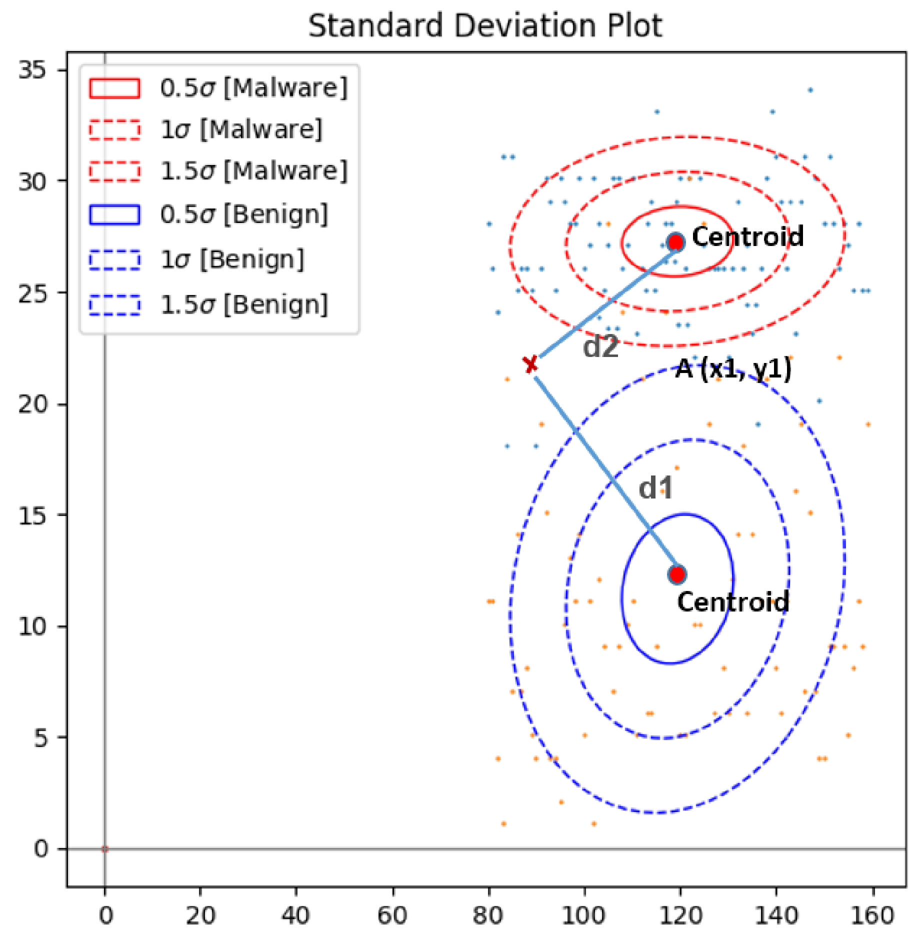
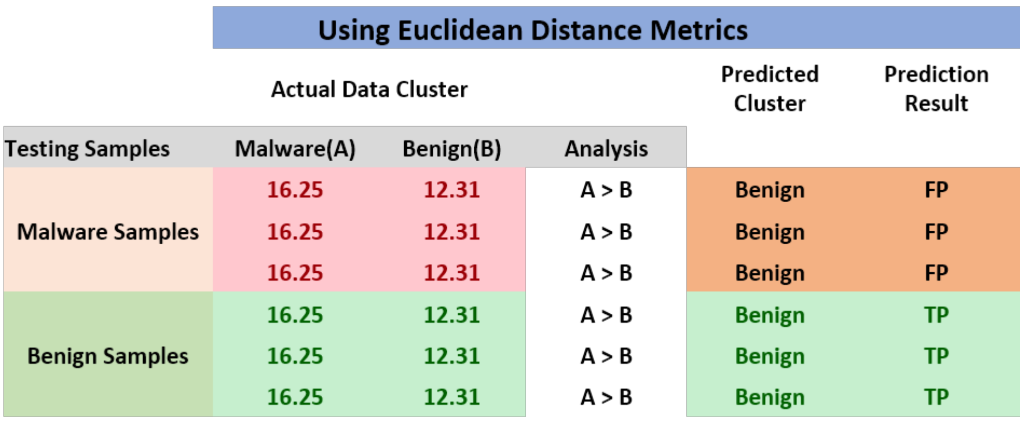


Figure 3. Mahalanobis distance calculation between two app clusters.

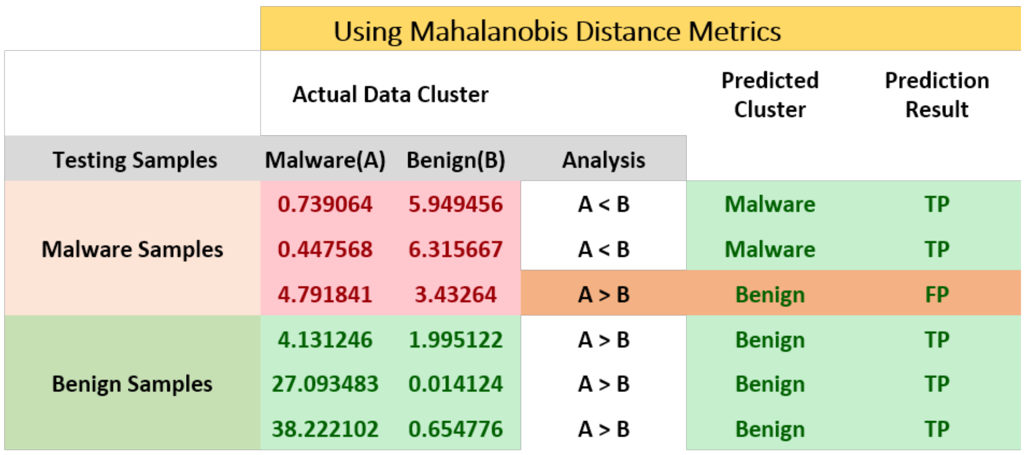


Using Euclidean Distance Metrics

Testing Samples	Malware(A)	Benign(B)	Analysis	Predicted Cluster	Prediction Result
	Actual Data Cluster				
Malware Samples	16.25	12.31	A > B	Benign	FP
	16.25	12.31	A > B	Benign	FP
	16.25	12.31	A > B	Benign	FP
Benign Samples	16.25	12.31	A > B	Benign	TP
	16.25	12.31	A > B	Benign	TP
	16.25	12.31	A > B	Benign	TP

Figure 4. Predicting the distance between apps using the Euclidean distance metric.

However, we can see that the Euclidean distance assumes the data are isotropically Gaussian, where the covariance matrix is assumed to be its identity. Thus, Euclidean distance treats all the given features equally. On the other hand, the Mahalanobis distance metric assumes the data are an anisotropic Gaussian distribution, whereby it measures the correlation between the variables. Figure 5 displays the results for six sample records; three were taken from each of the malware and benign datasets at random and tested for the Mahalanobis distance. The results show that 100% of the prediction was correct. For example, calculating the Mahalanobis distance for the first sample test record from the malware dataset, we can see that the distance from malware was 0.739, which is less than the Mahalanobis distance taken from the benign cluster (i.e., 5.95). Thus, we can classify the new data as malware. On the other hand, malware sample record 3 was misclassified as benign.



Using Mahalanobis Distance Metrics

	Actual Data Cluster			Predicted Cluster	Prediction Result
Testing Samples	Malware(A)	Benign(B)	Analysis		
Malware Samples	0.739064	5.949456	A < B	Malware	TP
	0.447568	6.315667	A < B	Malware	TP
	4.791841	3.43264	A > B	Benign	FP
Benign Samples	4.131246	1.995122	A > B	Benign	TP
	27.093483	0.014124	A > B	Benign	TP
	38.222102	0.654776	A > B	Benign	TP

Figure 5. Prediction of distance between apps using the Mahalanobis distance metric.

Reasoning on Dataset: From our study on the PI vulnerabilities, we found that features such as unsafePICreation, unsafeBroadcast, unsafePITransfer, and apps' dangerous permissions were highly correlated by directly enumerating the apps' vulnerable characters. Similarly, clustering the dataset using Mahalanobis distance showed a better performance compared with clustering based on the Euclidean distance metric.

In Algorithm 2, we explain the overall methodology, starting with automatically extracting the features from the given dataset and the calculation of the Mahalanobis distance to training the model and finally predicting the new data points using the above trained model.

Algorithm 2 Training and prediction using Mahalanobis distance

```
Data: CSV Dataset
Result: Model (to predict using Mahalanobis distance)
Fitness, Subsets ← Apply evolutionary selection method (Gnerations = 20)
BestFeature ← Top(Fitness)
HopkinsValue ← HopkinsStatistics(Dataset)
if HopkinsValue > 0.5 then
    xtrain, xtest, ytrain, ytest ← Training + Testing Data Split
    Distance[] ← Mahalanobis(source = Dataset, features = BestFeatures)
    #Test Dataset
    MahalanobisBinaryClassifier(xtrain, ytrain)
    auc-roc ← roc_auc_score(truth, scores)   #AUC-ROC Curve
    TP, TN, FP, FN ← confusion_matrix(truth, pred)   #Confusion Matrix
    accuracy ← accuracy_score(truth, pred)   #Accuracy of the Model
end
H < 0.5, Dataset doesn't have statistically significant clusters.
```

4. Evaluation

The proposed work, MULBER, was developed using Java, the dex parser was implemented using ANTLR, Weka was used for the evolutionary feature selection algorithm, and the clustering of records based on the Mahalanobis distance metric was implemented in Python.

4.1. Research Objectives

RO-1: To investigate automatic feature selection related to PendingIntent vulnerabilities, implicit broadcast, and usage of dangerous permissions.
RO-2: To investigate the impact of the proposed algorithm on the performance of Android malware clustering.
RO-3: To evaluate the detection capability of the proposed model with the RAICC dataset [13] containing Android applications with PendingIntent vulnerabilities.

4.2. Dataset Generation

We tested MULBER on the following dataset and classified our dataset into three types. The first dataset consisted of the CICMalDroid-2020 dataset [52], a collection of 17,241 Android app samples spanning five distinct categories: adware, banking malware, SMS malware, riskware, and benign, and extensive samples from several sources including the VirusTotal service, Contagio security blog, AMD, MalDozer, and other datasets used by recent research contributions. From the dataset, we extracted the following statistical information: (1) PendingIntent flags; (2) usage of public broadcast; and (3) exchange of PendingIntent using implicit Intent (as displayed in Table 2).

4.2.1. Classification of APKs Using PI Vulnerabilities

Report

Based on our study of the 22,638 apps (i.e., a collection of $>$15 million classes composed of $>$70 million methods), we found the following interesting highlights:

Highlights from Table 2:

1. $\sim$9% of PI vulnerabilities were found in benign-tagged apps taken from the CICMalDroid-2020 dataset [52].
2. $\sim$1.4% of implicit Intent vulnerabilities were found in benign apps taken from the CICMalDroid-2020 dataset.
3. $\sim$0.15% of apps had vulnerable PI transfers (though it looks trivial, it can create precarious behavior in apps).
4. In total, MULBER identified, $\sim$26% of dataset apps that had PendingIntent- and Intent-based vulnerabilities.

The above results show that the dataset had PI-based vulnerabilities even in the benign applications. By considering a PI-based vulnerability a security feature, our diagnosis exposes a new security vulnerability that needs attention in app classifications.

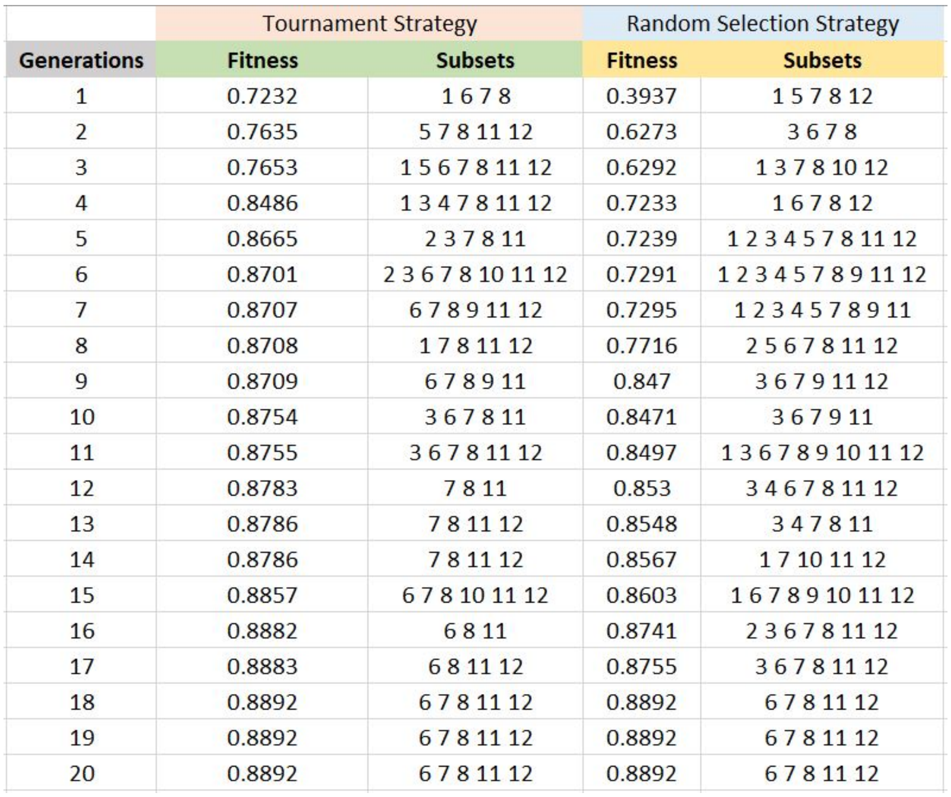
4.3. Evolutionary Feature Selection [RO-1]

MULBER applied an evolutionary feature selection algorithm to automatically extract the best features from the given dataset. We tested the feature selection using the tournament selection operator. In the tournament selection strategy, the individuals are selected from the outcomes of several tournaments. The tournament winner is named the best candidate and is selected for the crossover. The results are given in the following Figure 6.

Table 2. Evaluation using the CICMalDroid-2020 and Drebin datasets.

Dataset	**#App**	**#C**	**#M**	**#I**	**#PI**	**1s**	**Nc**	**Cc**	**Uc**	**Im**	***PubBr***	***VulPI***	***VulTR***	**%VulIn**	**%VulPI**	**%VulTR**
Drebin	5546	1,338,542	7,174,900	168,444	16,072	367	92	3570	2056	12	963	3916	87	0.57	24.3653559	0.541314087
SMS	4796	160,287	794,577	61,700	14,044	2	1	6094	303	0	771	7755	0	1.249594814	55.21931074	0
Adware	1510	584,945	3,331,055	99,125	11,894	43	23	2573	4197	119	703	957	8	0.709205549	8.046073651	0.067260804
Riskware	4352	2,464,826	12,713,338	145,208	14,753	911	40	841	6101	13	1312	4380	5	0.903531486	29.68887684	0.033891412
Banking	2399	673,494	3,452,858	63,421	7157	824	13	578	1776	3	556	2489	26	0.876681225	34.77714126	0.363280704
Benign	4035	10,229,928	46,168,467	476,672	25,410	1662	502	3154	10,089	58	6570	2264	322	1.378306257	8.909878001	1.267217631
Total	22,638	15,452,022	73,635,195	1,014,570	89,330	3809	671	16,810	24,522	205	10,875	21,761	448	1.07188267	24.36023732	0.50151125

NOTE: 1s—FLAG_ONE_SHOT; Nc—FLAG_NO_CREATE; Cc—FLAG_CANCEL_CURRENT; Uc—FLAG_UPDATE_CURRENT; Im— FLAG_IMMUTABLE; PI—PendingIntent; C—Class; $PubBr$—Public Broadcast; $VulPI$—Vulnerable PI Creation; I—Intent; $VulTR$—Weak PI Transfer; %$VulPI$—Weak PI Creation (%); M—Method; %$VulTR$—Weak PI Transfer (%); %$VulIm$—Implicit Intent (%).



	Tournament Strategy		Random Selection Strategy	
Generations	Fitness	Subsets	Fitness	Subsets
1	0.7232	1 6 7 8	0.3937	1 5 7 8 12
2	0.7635	5 7 8 11 12	0.6273	3 6 7 8
3	0.7653	1 5 6 7 8 11 12	0.6292	1 3 7 8 10 12
4	0.8486	1 3 4 7 8 11 12	0.7233	1 6 7 8 12
5	0.8665	2 3 7 8 11	0.7239	1 2 3 4 5 7 8 11 12
6	0.8701	2 3 6 7 8 10 11 12	0.7291	1 2 3 4 5 7 8 9 11 12
7	0.8707	6 7 8 9 11 12	0.7295	1 2 3 4 5 7 8 9 11
8	0.8708	1 7 8 11 12	0.7716	2 5 6 7 8 11 12
9	0.8709	6 7 8 9 11	0.847	3 6 7 9 11 12
10	0.8754	3 6 7 8 11	0.8471	3 6 7 9 11
11	0.8755	3 6 7 8 11 12	0.8497	1 3 6 7 8 9 10 11 12
12	0.8783	7 8 11	0.853	3 4 6 7 8 11 12
13	0.8786	7 8 11 12	0.8548	3 4 7 8 11
14	0.8786	7 8 11 12	0.8567	1 7 10 11 12
15	0.8857	6 7 8 10 11 12	0.8603	1 6 7 8 9 10 11 12
16	0.8882	6 8 11	0.8741	2 3 6 7 8 11 12
17	0.8883	6 8 11 12	0.8755	3 6 7 8 11 12
18	0.8892	6 7 8 11 12	0.8892	6 7 8 11 12
19	0.8892	6 7 8 11 12	0.8892	6 7 8 11 12
20	0.8892	6 7 8 11 12	0.8892	6 7 8 11 12

Figure 6. Evolutionary feature selection—comparison of tournament and random selection strategies based on 20 generations.

The first generation population consisted of individuals representing the dataset columns {1, 6, 7, and 8} (i.e., {flag1s, unsafePICreation, unsafeBroadcast, unsafePITransfer}) and {1, 5, 7, 8, and 12} represented ({flag1s, flagim, unsafeBroadcast, unsafePITransfer, unknownOrdeprecatedPermsCount}) based on the tournament and random selection strategies. The initial population did not contain the complete knowledge that was required to represent the PI vulnerability (for example, an application's PI leak, implicit broadcast, or usage of dangerous permissions). Over the generations, we obtained the best fitness value as 0.8892, with a subset containing the following individual records {6, 7, 8, 11, and 12} representing the columns {unsafePICreation, unsafeBroadcast, unsafePITransfer, dangerousPermsCount, customOrdeprecatedPermsCount, and isDangerousPermissionsUsed}. From the above results, we can see that all the required information related to PI vulnerabilities was successfully extracted, for example, the presence of unsafePICreation, unsafe PITransfer, and dangerous permissions (greater than 0) can indicate that the application under investigation had PI vulnerabilities.

Thus, we can conclude that our system can automatically learn the required features related to PI vulnerabilities through the evolutionary feature selection mechanism.

4.4. Clustering Using Mahalanobis [RO-2]

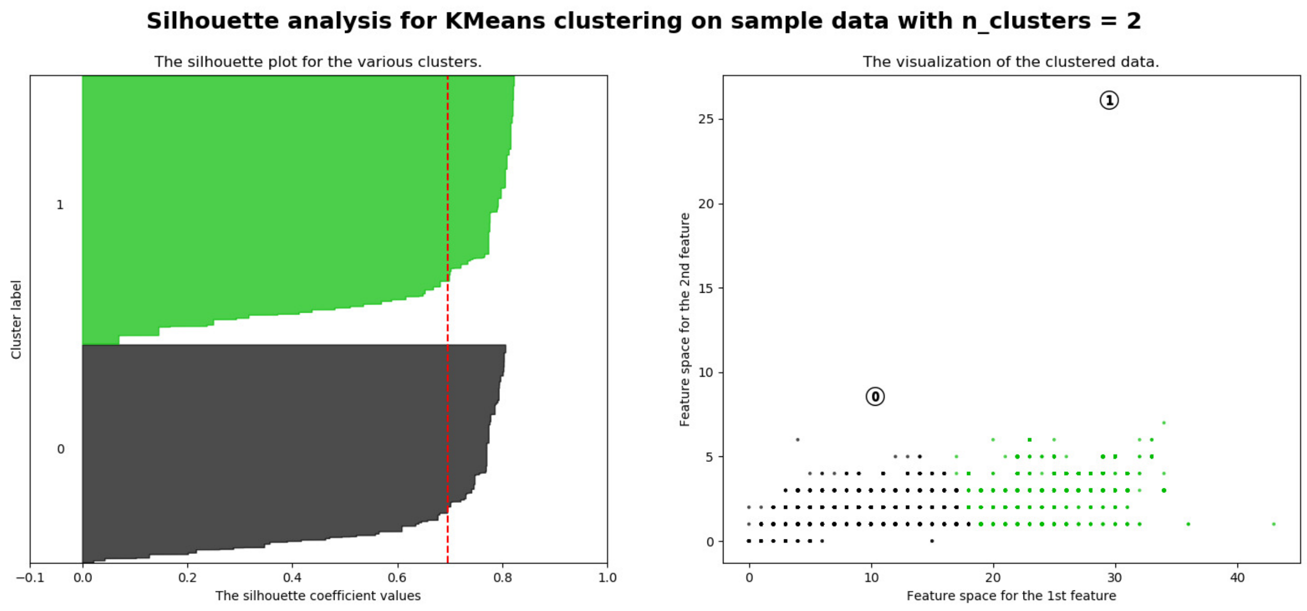
MULBER applied the evolutionary feature selection algorithm to automatically extract the best feature from the given dataset. We tested the feature selection using the tournament selection operator. In the tournament selection strategy, the individuals are selected from the outcomes of several tournaments. The tournament winner is named the best candidate, and is selected for the crossover. Following evolutionary feature selection, MULBER analyzed the data to check if the generated set is clusterable using Hopkins statistics [53] and we obtained a value close to 1, showing that it was highly clustered.

$$\text{Hopkins statistics (H)} = \frac{\sum_{i=1}^{n} u_i^d}{\sum_{i=1}^{m} u_i^d + \sum_{i=1}^{m} w_i^d} \tag{12}$$
$$\text{H} = 0.9986$$

Merged The Equation (Hopkins Statistics (12)).

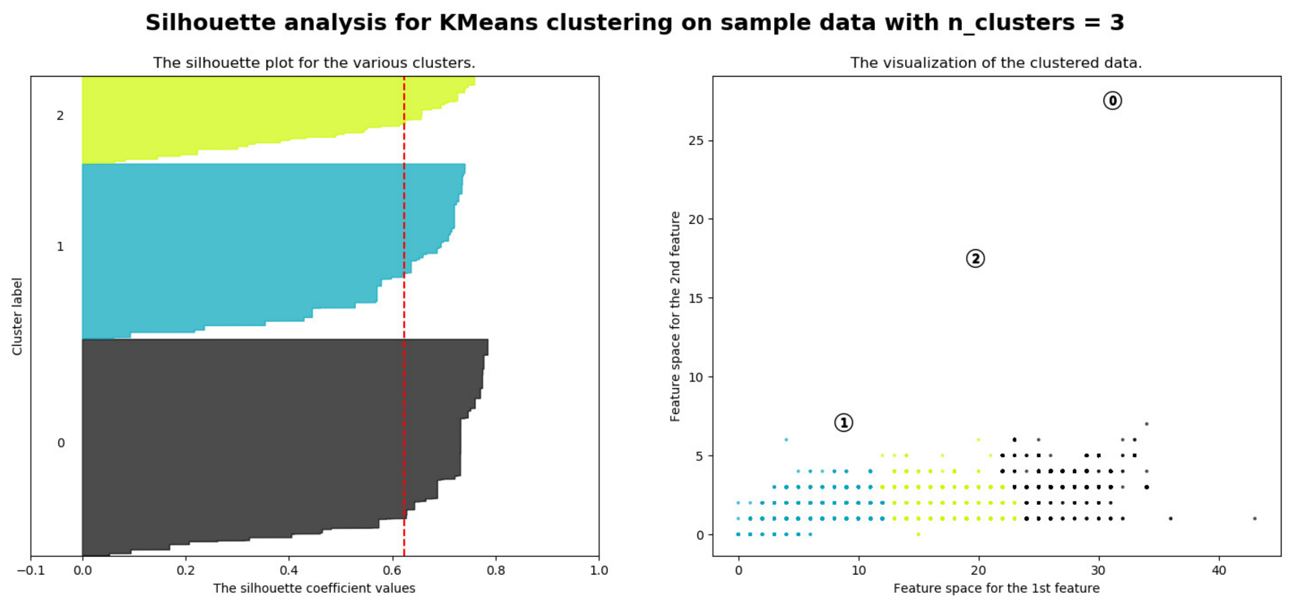
Following the Hopkins statistics, MULBER identified the number of possible clusters that can be formed from the generated dataset using the silhouette coefficient. The silhouette coefficient value was between [−1, 1], where a score of 1 was the best value and represented very compact data points within the cluster, −1 represented the worst value, and when the value was close to 0, it denoted overlapping clusters. Our silhouette analysis for the number of clusters that can be formed is given in Figure 7. Based on the silhouette coefficient value, we can say that by using n_cluster = 2, we could obtain points that were closer within the cluster and further away from the other cluster points. Figure 8 shows the distribution and the relationships between the variables from the perspective of the static features of the application.

Following the silhouette coefficient, MULBER also tested the generated dataset's correlation matrix, as given in Figure 9, for the different features selected using the evolutionary selection model. The matrix depicts the correlation between all the possible pairs of features. Finally, the outliers in our dataset were identified, as shown in Figure 10, and the data identified as having no outliers can help to train our model and predict the expected results. Finally, we applied the Mahanalobis binary classifier to the clean dataset. The ROC curve is given in the Figure 11.



(**a**) 2 cluster, Score = 0.69

Figure 7. *Cont.*



(**b**) 3 cluster, Score = 0.62

Figure 7. Sillhoutte coefficient identified using MULBER.

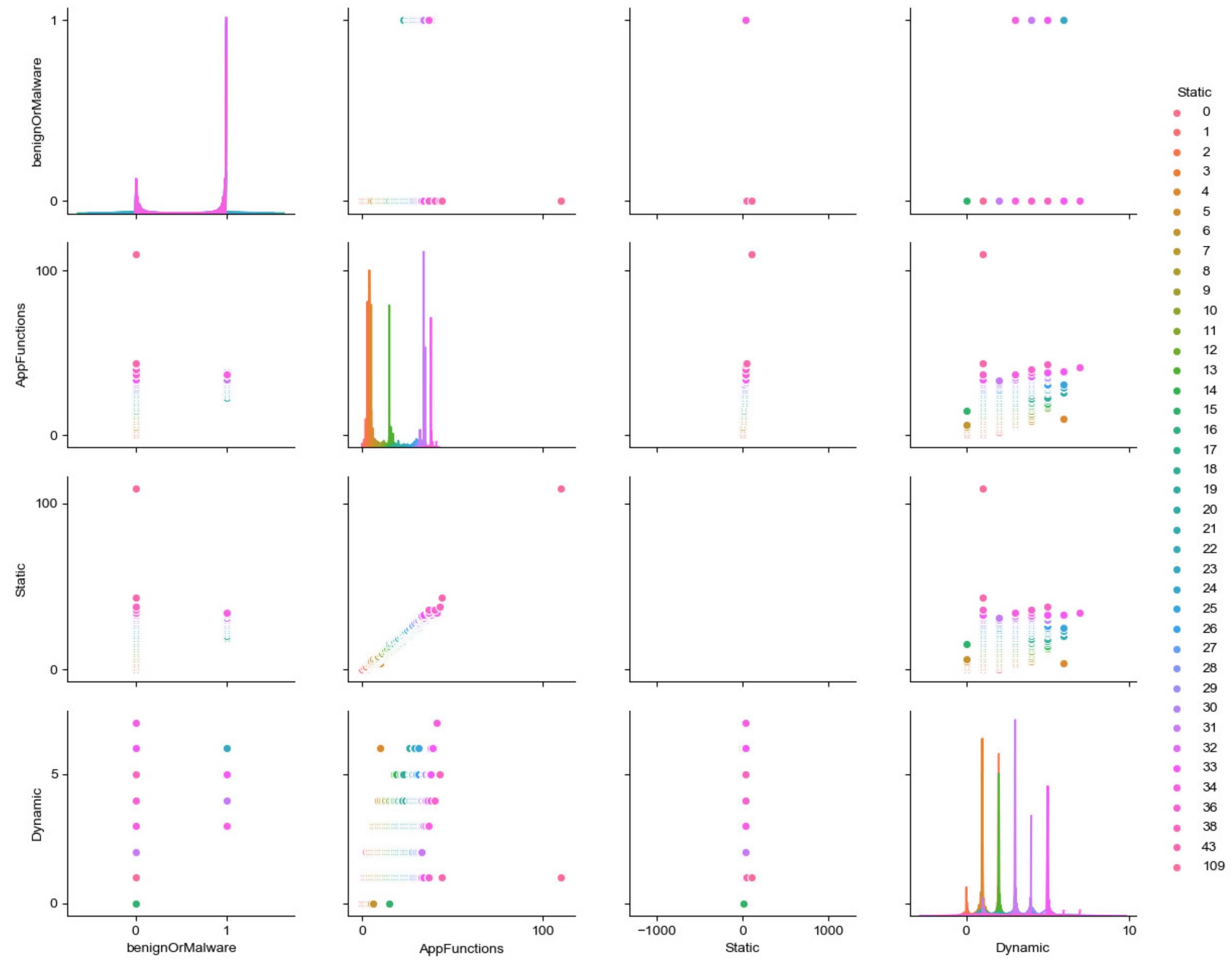


Figure 8. Exploratory data analysis (EDA) using pair plot.

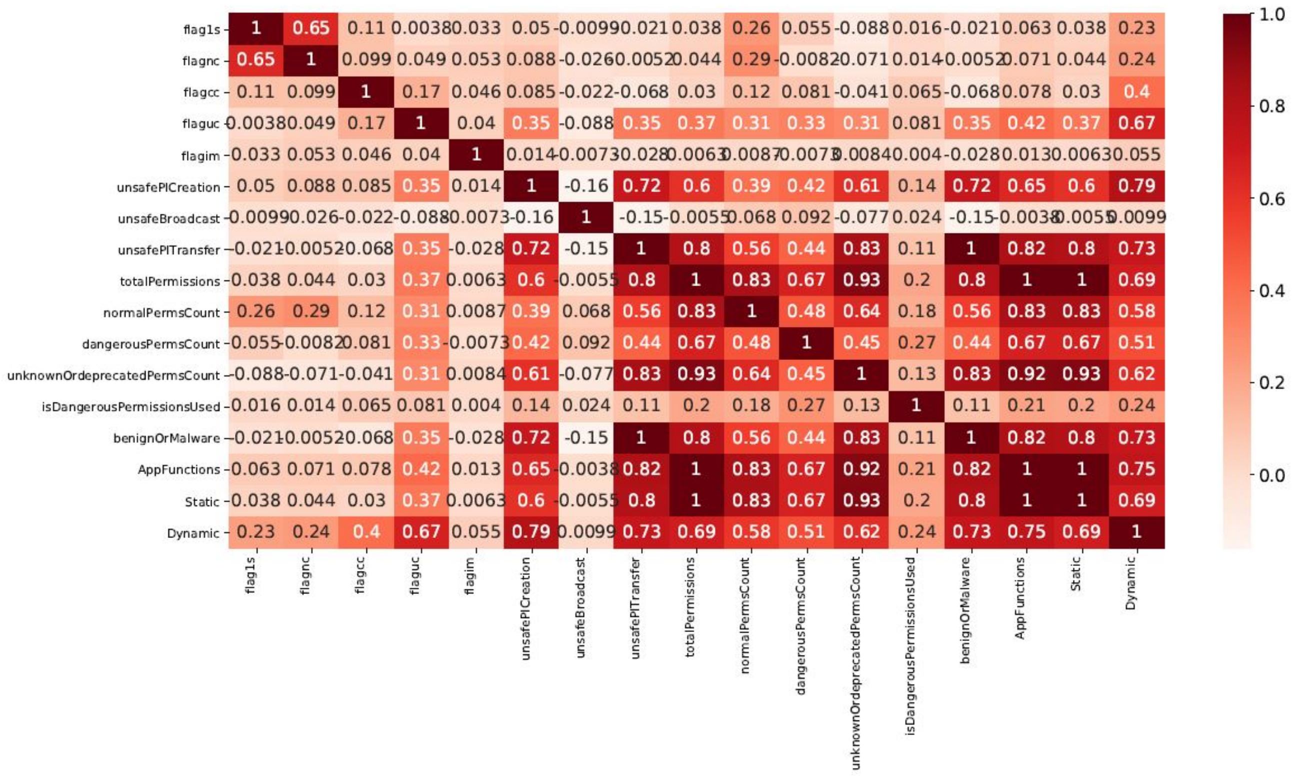


Figure 9. Correlation matrix calculated for the features extracted from the evolutionary feature selection.

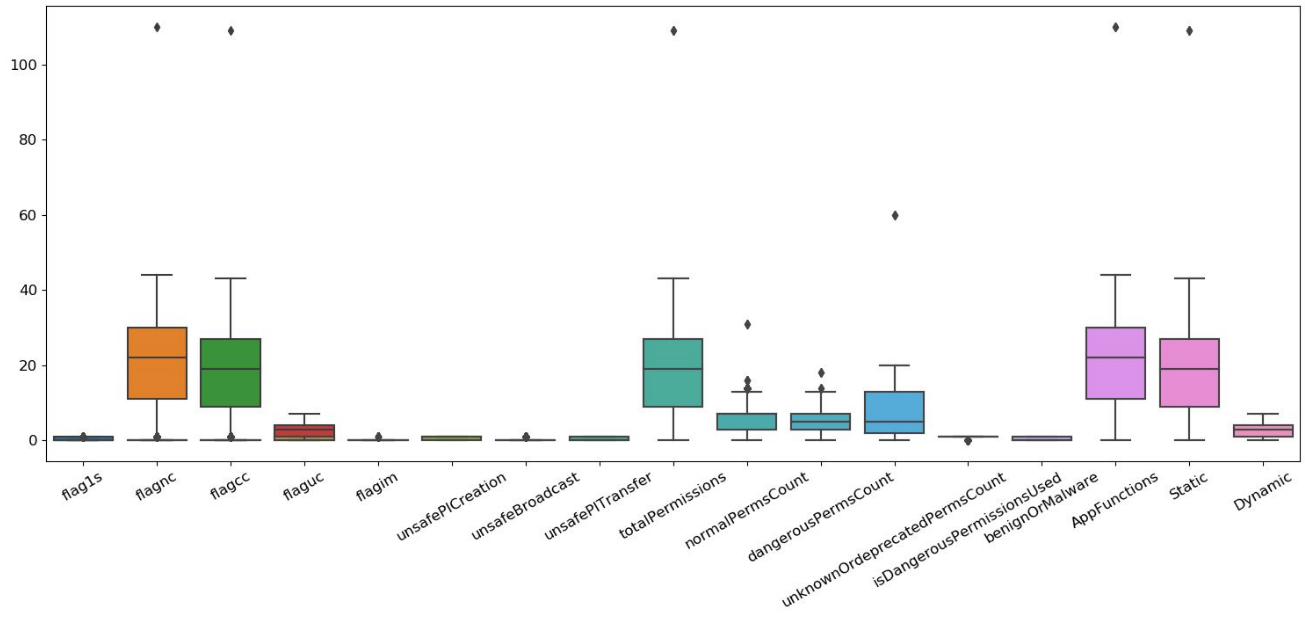


Figure 10. Box plot to identify the outlier in the dataset.

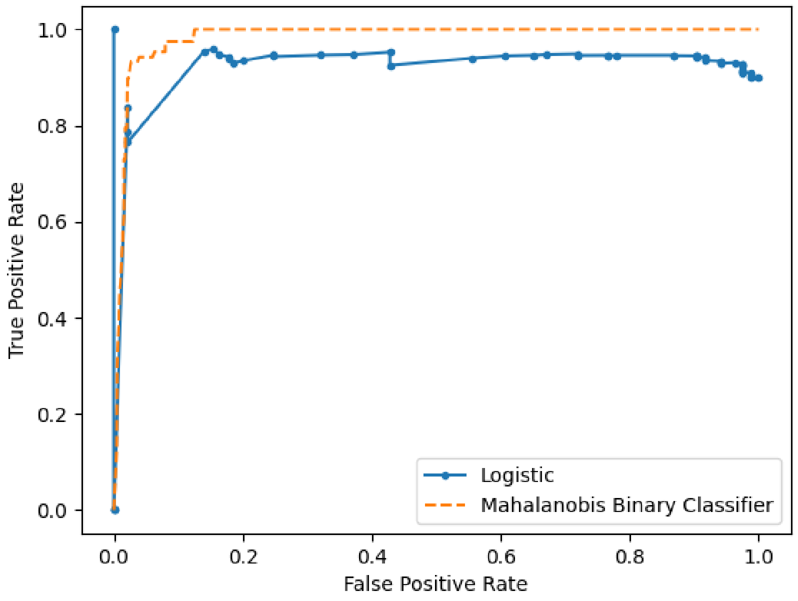
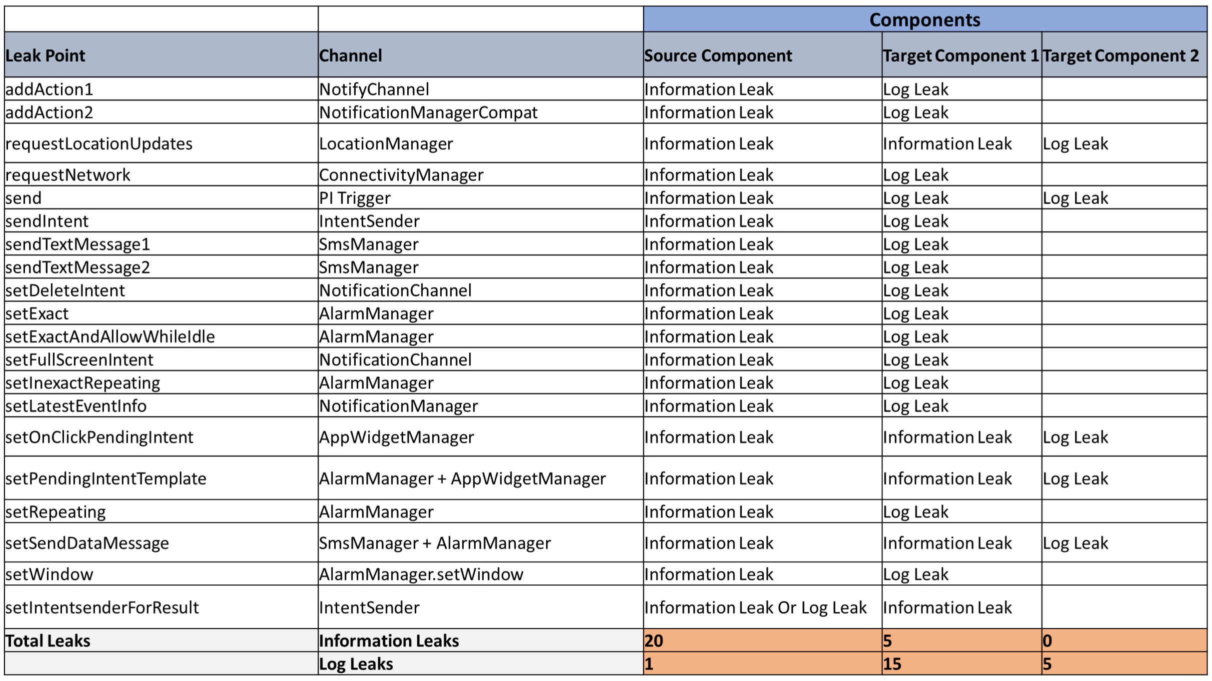


Figure 11. AUC-ROC curve comparing the Mahalanobis binary classifiers.

4.5. Evaluation Using RAICC Dataset [RO3]

We compared MULBER with the well-known RAICC dataset that consists of 20 applications demonstrating the various possible means of PI vulnerabilities, as given in Figure 12. The RAICC dataset enumerates two kinds of vulnerabilities: (1) information leaks, and (2) log leaks. In information leak vulnerabilities, sensitive information is leaked during inter-component communication scenarios using Intent and PendingIntent. In log leak vulnerabilities, sensitive information is leaked using the application logging mechanism (a side-channel attack).



		Components		
Leak Point	**Channel**	**Source Component**	**Target Component 1**	**Target Component 2**
addAction1	NotifyChannel	Information Leak	Log Leak	
addAction2	NotificationManagerCompat	Information Leak	Log Leak	
requestLocationUpdates	LocationManager	Information Leak	Information Leak	Log Leak
requestNetwork	ConnectivityManager	Information Leak	Log Leak	
send	PI Trigger	Information Leak	Log Leak	Log Leak
sendIntent	IntentSender	Information Leak	Log Leak	
sendTextMessage1	SmsManager	Information Leak	Log Leak	
sendTextMessage2	SmsManager	Information Leak	Log Leak	
setDeleteIntent	NotificationChannel	Information Leak	Log Leak	
setExact	AlarmManager	Information Leak	Log Leak	
setExactAndAllowWhileIdle	AlarmManager	Information Leak	Log Leak	
setFullScreenIntent	NotificationChannel	Information Leak	Log Leak	
setInexactRepeating	AlarmManager	Information Leak	Log Leak	
setLatestEventInfo	NotificationManager	Information Leak	Log Leak	
setOnClickPendingIntent	AppWidgetManager	Information Leak	Information Leak	Log Leak
setPendingIntentTemplate	AlarmManager + AppWidgetManager	Information Leak	Information Leak	Log Leak
setRepeating	AlarmManager	Information Leak	Log Leak	
setSendDataMessage	SmsManager + AlarmManager	Information Leak	Information Leak	Log Leak
setWindow	AlarmManager.setWindow	Information Leak	Log Leak	
setIntentsenderForResult	IntentSender	Information Leak Or Log Leak	Information Leak	
Total Leaks	**Information Leaks**	**20**	**5**	**0**
	Log Leaks	**1**	**15**	**5**

Figure 12. RAICC dataset showing multiple PI leaks.

We evaluated MULBER using the following three metrics: (1) precision, (2) recall, and (3) F1-score.

Precision: The precision metric determines how many of the exactly predicted values from the overall positive classes are actually positive. The precision value is anticipated when the number of false positives is minimal.

$$Precision(p) = \frac{TP}{TP + FP} \tag{13}$$

Recall (or) Sensitivity: The recall metric finds the accurately predicted values from all the positive classes. It is also called be true positive rate (TPR). The recall value is much more desirable when the number of false negatives is minimal.

$$Recall(r) = \frac{TP}{TP + FN} \tag{14}$$

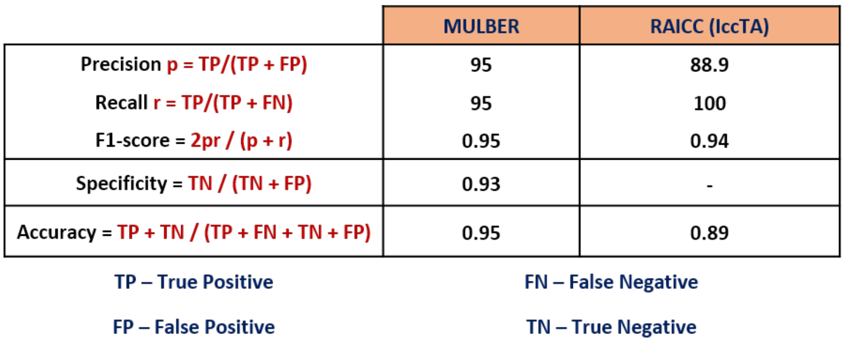
F1-score: F1-score or F1 measure is the equilibrium between the precision and recall metrics. It is considered the most effective metric for quantifying real-world problems, since it mostly contains imbalanced class distributions.

$$F1 - score = 2 * \frac{pr}{p + r} \tag{15}$$

Specificity: Specificity is the ratio of true negatives identified to total negatives present in the data.

$$Specificity = \frac{TN}{TN + FP} \tag{16}$$

Figure 13 shows that MULBER was better at detecting PI-based vulnerabilities; however, compared to RAICC, MULBER had a lower recall value. Overall, from the F1 measure, we can see that MULBER was better at identifying vulnerabilities based on PendingIntent leaks. The results also show high specificity, which implies that MULBER was better at identifying apps without PI vulnerabilities, thereby had fewer false positives.



	MULBER	RAICC (IccTA)
Precision p = TP/(TP + FP)	95	88.9
Recall r = TP/(TP + FN)	95	100
F1-score = 2pr / (p + r)	0.95	0.94
Specificity = TN / (TN + FP)	0.93	-
Accuracy = TP + TN / (TP + FN + TN + FP)	0.95	0.89

TP – True Positive FN – False Negative

FP – False Positive TN – True Negative

Figure 13. Evaluation results of MULBER vs. RAICC.

5. Limitations

This paper proposes the first methodology to automate the process of extracting features from an APK by performing direct binary analysis for PendingIntent and Intent vulnerabilities, identifying the best-fit features to consider, and performing the Mahalanobis distance calculation to classify an app's behavior.

In this paper, we did not consider some properties such as the following:

1. Imbalanced Dataset—This property was not considered in this paper, as training a model using a balanced dataset improves the accuracy; this paper focused mainly on the mechanism for handling the mutivariate features of applications using the Mahalanobis distance metric rather than Euclidean-metric-based clustering.
2. This paper considered the PendingIntent and Intent exchanges between applications; however, as a future work, we plan to extend this to application behavior and other control-flow graph properties.

6. Future Work

In this paper, we used the Mahalanobis distance metric to classify applications as benign or malware. In the future, we plan to investigate other clustering methodologies such as the radial basis function (RBF) and fuzzy clustering method, where an app can belong to more than one class. For example, an app shows a sign of being benign, but internally performs malicious functionalities. In this paper, we did not consider the fuzzy clustering model since in order to consider the overlapping nature of an app, we need to perform additional control-flow and data-flow graph analyses. The current paper targeted features such as app capability and communication patterns. However, we plan to extend this work in the future to incorporate and analyze the fuzzy nature of Android applications.

7. Conclusions

This paper studied the malware clustering property of Android applications based on PendingIntent- and Intent-based communication properties using a Mahalanobis distance metric rather than following the regular Euclidean distance metric. The main challenge involved in handling this implicit Intent was the difficulty in identifying the participants, and the challenge involved in PendingIntent-based communications was that if the participant was malware, he could own the privileges of the PendingIntent creator. In this paper, we demonstrated the necessity of clustering malware apps using multivariate analysis since malware is the property of an app classified based on multiple dependent variables rather than single features, resulting in an app being classified as benign or malware. For example, classifying an app as malware based only on ICC communications is not accurate, rather there are multiple other factors that should be considered such as permissions held by the collaborating apps, the flag used to create the PendingIntent, etc. Finally, this paper proposed a novel framework called MULBER that completely automated the process of extracting features from dex code, selecting the best-fit dependent features using an evolutionary feature selection model, and clustering them using the Mahalanobis distance metric. Our model has proved to have good accuracy and an F1 score (0.95, 0.95) compared to RAICC with a (0.89, 0.94) score.

Author Contributions: Conceptualization, P.D.S.; methodology, P.D.S., G.S.; software, P.D.S.; validation, P.D.S., G.S.; formal analysis, P.D.S.; investigation, P.D.S., G.S.; resources, P.D.S., G.S.; data curation, P.D.S.; writing—original draft preparation, P.D.S.; writing—review and editing, P.D.S., G.S.; visualization, P.D.S.; supervision, G.S.; project administration, G.S.; funding acquisition, G.S. All authors have read and agreed to the published version of the manuscript.

Funding: VIT, University, Chennai, Tamil Nadu, India

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mobile Operating System Market Share Worldwide (2021–2022). Available online: https://gs.statcounter.com/os-market-share/mobile/worldwide/ (accessed on 3 October 2022).
2. Mobile Malware Evolution 2021. Available online: https://securelist.com/mobile-malware-evolution-2021/105876/ (accessed on 1 September 2022).
3. Intents and Intent Filters. Available online: https://developer.android.com/guide/components/intents-filters (accessed on 5 October 2022).
4. Enck, W.; Ongtang, M.; McDaniel, P. On lightweight mobile phone application certification. In Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 9–13 November 2009; pp. 235–245.
5. Felt, A.P.; Chin, E.; Hanna, S.; Song, D.; Wagner, D. Android permissions demystified. In Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11), Association for Computing Machinery, New York, NY, USA, 17–21 October 2011; pp. 627–638. [CrossRef]
6. Grace, M.; Zhou, Y.; Zhang, Q.; Zou, S.; Jiang, X. RiskRanker: Scalable and accurate zero-day android malware detection. In Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12), Association for Computing Machinery, New York, NY, USA, 10–13 November 2012; pp. 281–294. [CrossRef]

7. CVE-2021-25352. Samsung Mobile. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25352 (accessed on 1 September 2022).
8. CVE-2021-25364. Samsung Mobile. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-25364 (accessed on 1 September 2022).
9. CVE-2020-7039. Samsung Mobile. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7039 (accessed on 1 September 2022).
10. CVE-2022-22286. Samsung Mobile. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22286 (accessed on 1 September 2022).
11. Mahalanobis, P.C. On the generalised distance in statistics (PDF). *Proc. Natl. Inst. Sci. India* **1936**, *2*, 49–55.
12. Euclidean Distance. Available online: https://en.wikipedia.org/wiki/Euclidean_distance (accessed on 5 October 2022).
13. Samhi, J.; Bartel, A.; Bissyandé, T.F.; Klein, J. Raicc: Revealing atypical inter-component communication in android apps. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 25–28 May 2021; pp. 1398–1409.
14. Groß, S.; Tiwari, A.; Hammer, C. Pianalyzer: A precise approach for pendingintent vulnerability analysis. In *European Symposium on Research in Computer Security*; Springer: Cham, Switzerland, 2018; pp. 41–59.
15. Li, L.; Bartel, A.; Bissyandé, T.F.; Klein, J.; Le Traon, Y.; Arzt, S.; Rasthofer, S.; Bodden, E.; Octeau, D.; McDaniel, P. Iccta: Detecting inter-component privacy leaks in android apps. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 280–291.
16. Wei, F.; Roy, S.; Ou, X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *Acm Trans. Priv. Secur.* **2014**, *21*, 1–32. [CrossRef]
17. Duraisamy, S.P.; Geetha, S.; Cheng, X.; Kadry, S. On Shielding Android's Pending Intent from Malware Apps Using a Novel Ownership-Based Authentication. *J. Circuits Syst. Comput.* **2022**, *31*, 13.
18. Zhang, C.; Li, S.; Diao, W.; Guo, S. PITracker: Detecting Android PendingIntent Vulnerabilities through Intent Flow Analysis. In Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '22), Association for Computing Machinery, New York, NY, USA, 20–25 May 2022. [CrossRef]
19. Duraisamy Soundrapandian, P.K.; Bao, T.; Baek, J.; Shoshitaishvili, Y.; Doupé, A.; Wang, R.; Ahn, G.J. Mutent: Dynamic android intent protection with ownership-based key distribution and security contracts. In Proceedings of the 54th Hawaii International Conference on System Sciences, Kauai, HI, USA, 5–8 January 2021; pp. 7217–7226.
20. Stone, M. Securing the System: A Deep Dive into Reversing Android Pre-Installed Apps. Available online: https://i.blackhat.com/USA-19/Thursday/us-19-Stone-Securing-The-System-A-Deep-Dive-Into-Reversing-Android-Preinstalled-Apps.pdf (accessed on 1 September 2022).
21. Lee, Y.K.; Bang, J.Y.; Safi, G.; Shahbazian, A.; Zhao, Y.; Medvidovic, N. A sealant for inter-app security holes in android. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering, Buenos Aires, Argentina, 20–28 May 2017; pp. 312–323.
22. Hammad, M.; Garcia, J.; Malek, S. Self-protection of android systems from inter-component communication attacks. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 726–737.
23. Hammad, M.; Garcia, J.; Malek, S. Static analysis of implicit control flow: Resolving java reflection and android intents. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2015.
24. Alhanahnah, M.; Yan, Q.; Bagheri, H.; Zhou, H.; Tsutano, Y.; Srisa-An, W.; Luo, X. DINA: Detecting Hidden Android Inter-App Communication in Dynamic Loaded Code. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 2782–2797. [CrossRef]
25. Demissie, B.F.; Ceccato, M.; Shar, L.K. Security analysis of permission re-delegation vulnerabilities in android apps. *Empir. Softw. Eng.* **2020**, *25*, 5084–5136. [CrossRef]
26. Felt, A.P.; Wang, H.J.; Moshchuk, A.; Hanna, S.; Chin, E. Permission re-delegation: Attacks and defenses. *Usenix Symp.* **2011**, *22*, 88.
27. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Traon, Y.L.; Octeau, D.; McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Not.* **2014**, *49*, 259–269. [CrossRef]
28. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing inter-application communication in android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 28 June–1 July 2011; pp. 239–252.
29. Lu, L.; Li, Z.; Wu, Z.; Lee, W.; Jiang, G. Chex: Statically vetting android apps for component hijacking vulnerabilities. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 229–240.
30. Demissie, B.F.; Ceccato, M.; Shar, L.K. Anflo: Detecting anomalous sensitive information flows in android apps. In Proceedings of the 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Gothenburg, Sweden, 27 May–3 June 2018.
31. Sadeghi, A.; Jabbarvand, R.; Ghorbani, N.; Bagheri, H.; Malek, S. A temporal permission analysis and enforcement framework for android. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 846–857.

32. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C.E.R.T. Drebin: Effective and explainable detection of android malware in your pocket. *Ndss* **2014**, *14*, 23–26.
33. Institute for System Security—Technische Universität Braunschweig. The Drebin Dataset. Available online: https://www.sec.cs.tu-bs.de/~danarp/drebin/index.html (accessed on 1 September 2022).
34. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *Acm Trans. Comput. Syst.* **2010**, *32*, 393–407. [CrossRef]
35. Dawoud, A.; Bugiel, S. Droidcap: Os support for capability-based permissions in android. *Ndss Symp.* **2019**. Available online: https://svenbugiel.github.io/publication/dawoud-19-ndss/dawoud-19-ndss.pdf (accessed on 5 October 2022)
36. Xu, Y.; Witchel, E. Maxoid: Transparently confining mobile applications with custom views of state. In Proceedings of the Tenth European Conference on Computer Systems, New York, NY, USA, 21–24 April 2015; pp. 1–16.
37. Jia, L.; Aljuraidan, J.; Fragkaki, E.; Bauer, L.; Stroucken, M.; Fukushima, K. Run-time enforcement of information-flow properties on android. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 775–792.
38. Rubio-Medrano, C.E.; Hill, M.; Claramunt, L.M.; Baek, J.; Ahn, G.J. DyPolDroid: Protecting Users and Organizations from Permission-Abuse Attacks in Android. In *International Conference on Secure Knowledge Management in Artificial Intelligence Era*; Krishnan, R., Rao, H.R., Sahay, S.K., Samtani, S., Zhao, Z., Eds.; SKM 2021; Springer: Cham, Switzerland, 2021; pp. 23–36. [CrossRef]
39. Conversational Actions. Intents. Available online: https://developers.google.com/assistant/conversational/intents (accessed on 5 October 2022).
40. Google Developers Guide. Pendingintent. Available online: https://developer.android.com/reference/android/app/PendingIntent (accessed on 1 September 2022).
41. Application Signing. Application Signing. Available online: https://source.android.com/security/apksigning#v1 (accessed on 1 September 2022).
42. Ben Gruver. Baksmali. (Disassembler for the Dex Format Used by Dalvik). Available online: https://github.com/JesusFreke/smali/tree/master/baksmali (accessed on 1 September 2022).
43. Ben Gruver. Smali. (Assembler for the Dex Format Used by Dalvik). Available online: https://github.com/JesusFreke/smali (accessed on 1 September 2022).
44. Ben Gruver. Dexlib2. Available online: https://github.com/JesusFreke/smali/tree/master/dexlib2 (accessed on 1 September 2022).
45. Tang, Y.; Pan, Z.; Pedrycz, W.; Ren, F.; Song, X. Viewpoint-Based Kernel Fuzzy Clustering With Weight Information Granules. *IEEE Trans. Emerg. Top. Comput. Intell.* **2022**. [CrossRef]
46. Gupta, N.; Ari, S.; Panigrahi, N. Change Detection in Landsat Images Using Unsupervised Learning and RBF-Based Clustering. *IEEE Trans. Emerg. Top. Comput. Intell.* **2021**, *5*, 284–297. [CrossRef]
47. Roseline, SA.; Geetha, S.; Kadry, S.; Nam, Y. Intelligent vision-based malware detection and classification using deep random forest paradigm. *IEEE Access.* **2020**, *8*, 206303-206324. [CrossRef]
48. Hemalatha, J.; Roseline, S.A.; Geetha, S.; Kadry, S.; Damaševičius, R. An Efficient DenseNet-Based Deep Learning Model for Malware Detection. *Entropy* **2021**, *23*, 344. [CrossRef] [PubMed]
49. Liberti, L.; Lavor, C.; Maculan, N.; Mucherino, A. Euclidean Distance Geometry and Applications. *SIAM Rev.* **2014**, *56*, 3–69. [CrossRef]
50. Ghorbani, H. Mahalanobis Distance and Its Application for Detecting Multivariate Outliers. *Facta Univ. Ser. Math. Inform.* **2019**, *34*, 583–595. [CrossRef]
51. Mahalanobis Distance. Available online: https://en.wikipedia.org/wiki/Mahalanobis_distance (accessed on 5 October 2022).
52. Canadian Institute for Cybersecurity. Available online: https://www.unb.ca/cic/datasets/maldroid-2020.html (accessed on 1 September 2022).
53. Hopkins Statistic. Available online: https://en.wikipedia.org/wiki/Hopkins_statistic/ (accessed on 1 September 2022).